

Problem Session 7: Optimization and Caching

SOLUTION

March 18, 2021

1. **Optimization.** You have just joined a new startup that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they converted to the code to use iteration:

```
int fact(int n){
    int i;
    int result = 1;

    for (i=n; i > 0; i--){
        result = result * i;
    }

    return result;
}
```

By doing so, they have reduced the number of cycles per element (CPE) for the function from around 63 to around 4 (really!). Still, they would like to do better.

- (a) One of the programmers heard about loop unrolling. He generated the following code:

```
int fact_u2(int n){
    int i;
    int result = 1;

    for (i = n; i > 0; i-=2){
        result = (result *i) * (i-1);
    }

    return result;
}
```

Is this a valid optimization that a compiler might perform? If so, justify why the two functions are equivalent. If not, state which values of n will return different values and show how to fix it.

No, optimization does not behave the same as original code. Will return 0 whenever n is odd.

```
int fact_u2(int n){
    int i;
```

```

int result = 1;

for (i = n; i > 1; i-=2){
    result = (result *i) * (i-1);
}

return result;
}

```

- (b) You modify the line inside the loop to read: `result = result * (i * (i-1))`; To everyone's astonishment, the measured performance now has a CPE of 2.5. How do you explain this improved performance?

The multiplication $i * (i-1)$ can overlap with the multiplication by result from the previous iteration.

- (c) Name two further changes might you make to try to further improve the performance of your factorial function.

unroll loop, separate accumulators

- (d) Show how to modify the code to improve the performance using the techniques identified in Part1c.

```

int fact_u2(int n){
    int result = 1;
    int result2 = 1;
    int result3 = 1;
    int result4 = 1;

    for (int i = n; i > 3; i-=4){
        result = result * i;
        result2 = result2 * (i-1);
        result3 = result3 * (i-2);
        result4 = result4 * (i-3);
    }
    result *= result2 * result3 * result4;

    for(;i > 0; i--){
        result *= i;
    }

    return result;
}

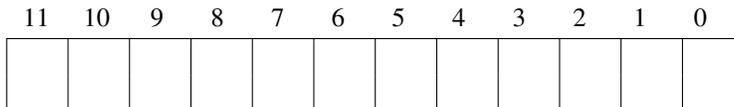
```

2. **Direct-Mapped Caches.** The following table depicts a direct-mapped cache, with an 8 byte block size and 4 cache lines:

Direct-Mapped Cache										
Index	Tag	Valid	Data							
0	29	0	34	29	8E	00	39	AE	AB	07
1	73	1	0D	8F	AA	E9	0C	3C	EA	01
2	A7	1	88	4B	E2	04	D2	13	B0	05
3	3B	1	AC	99	FF	1F	B5	47	0D	00

You should assume:

- Memory is byte addressable. All memory accesses read/write 1-byte.
 - Memory addresses are 12 bits.
- (a) The box below depicts a 12-bit memory address. Indicate (by labeling the diagram) the fields that would be used to determine (1) the tag, (2) the index, and (3) the offset.



CT: [11-5] CI: [4-3] CO: [2-0]

- (b) Consider the following sequence of accesses (yes, they occur sequentially). For each access, determine the tag, index, and offset. Then indicate whether that access would correspond to a cache hit or a cache miss, and what byte is read (if the exact value is unknown because it is not shown in the initial cache diagram, use the notation MEM[addr] instead of giving the byte).

Operation	Tag	Index	Offset	Hit?	Byte read
i. Read 0xAB8	0x55	3	0	Miss	Mem[0xAB8]
ii. Read 0xE68	0x73	1	0	Hit	0x0D
iii. Read 0x524	0x29	0	4	Miss	Mem[0x524]
iv. Read 0xE6C	0x73	1	4	Hit	0x0C
v. Read 0x526	0x29	0	4	Hit	Mem[0x526]
vi. Read 0x528	0x29	1	0	Miss	Mem[0x528]